Chapter 5

Debugging

Some errors are easily deduced from a debugger backtrace: Using the wrong variable in an index, etc. Others, particularly when the program's behavior is unexpected but does not result in a crash, is easier to debug with application-level debugging output.

5.1 A Debugging Mindset

Debugging is a process of problem solving. It's similar to the process that physicians use to diagnose illness: a combination of experience, reasoning, and tools.

Before you start debugging intensively, check the really easy stuff:

- `make clean`—sometimes you may have failed to recompile a particular bit of code. It happens more than you'd think.
- `check compiler warnings`—they're there to let you be lazy. Let them help you. *Always* use `-Wall`. Consider also using `-Wextra, -Wshadow, -Wunreachable-code`.

Next, avoid the "it's not my bug" syndrome. You'll see bugs that "can't happen!" (But it did—you have a core dump or error to prove it.) You may be tempted to say "the compiler's buggy!" (the OS, the course staff, etc.). It's possible, but it's very unlikely. The bug is most likely in your code, and it's even more likely to be in the new code you've just added.

As a result, some debugging is easy—you can identify it quickly by looking at the output of the program, the debugger output (a stack trace is very useful), and take a minute to think about what could have caused the problem. Oftentimes, this is sufficient to find the bug, or narrow it down to a very small number of possibilities. This is particularly true in the fairly common case when the bug is in a small amount of recently added code, so look there first. The `svn diff` command can come in very handy for reminding yourself what's changed! Sometimes you may overlook a "simple" change, or have forgotten something done at 3am the night before.

Some very useful steps in debugging, many stolen shamelessly from Kernighan & Pike:

- **Use a stack trace.** The stack trace might tell you exactly where the bug is, if you're lucky.
- **Examine the most recent change.** Bugs usually show up there.
- **Read the code carefully.** Before you start tweaking, read the code and *think* about it for a bit.

- **Explain your code to someone else.** Oftentimes when you read your own code, you'll read what you *expect* to see, not what's really there. Explain the code to anyone, even a pet rock. It helps.

Harder debugging is often approached as a process of hypothesis elimination:

- **Make the bug reproducible.**
- Think about the symptoms (crashing, incorrect result, etc.) of the bug. Look at (or remember) the code involved. Identify the possible and likely causes of these symptoms.
- Think about the correct behavior that you expected the program to exhibit, and identify your reasoning / assumpgions about what state or logic flow in the program would make it actually do so.
- Identify an experiment that you can perform that will most effectively narrow down the universe of possible causes of the bug and reasons that the program didn't behave as expected.
- Repeat.

Experiments you can perform include things like:

- Add consistency checks to find out where your assumptions about the program state went wrong. Is the list *really* sorted? Did the buffer *really* contain a full packet?
- Add debugging output to show the state of particular bits of the program, or to identify which parts of the program were being reached (correctly or incorrectly) before the crash.
- Remove parts of the code that may be contributing to the bug/crash/etc., to see if they're really responsible. Note that steps like this are a lot safer if you've recently committed your code, perhaps to a development branch.

Some of this depends on experience: over time, you'll have seen more common bugs and can identify the patterns by which they manifest themselves. But think about it like a binary search process: is the bug caused by one of these causes, or one of those? Can you perform an experiment to cut the field in half?

Making the bug easily reproducible is a very important first step. Imagine that your IRC server crashed when there were 20 clients connected and they'd each sent 1000s of lines of text. The situation that caused the bug tells you little about the reason the program failed. Does the bug still happen if you have only 10 clients? 5? If they send only 10 lines of code? Simplify the cause as much as possible; in its simplest verison, it may directly point out the bug!

Writing a log file is a great way to help debug. We'll talk about this a little more in Section 5.2. Being able to `grep` through the logfile or analyze it can make the debugging process much easier.

Using tools such as electric fence or system call tracers (below) can help identify particular bugs rapidly.

Finally, if a bug is really persistent, start writing things down. You'll save yourself repeating tests needlessly and will be more likely to cover the space of possibilities.

Once you've found a bug, think about two things:

1. Have I made this bug elsewhere in the code? Bugs that result from misunderstanding interfaces, etc., are likely to show up in multiple places. Do a quick check to proactively eliminate other bugs.

2. How can I avoid making this mistake in the future? You might be able to add test cases to automatically find them, add assertions to the code to detect if the bug happens again, use compiler warnings to automatically detect them, or change the way you write the code to make it impossible to make the mistake.

5.2 Good old printf done better: Debug macros

Instead of randomly throwing in and removing `printf`s while trying to track down bugs, consider using a bit more structure: debug macros that let you add debugging printfs and not remove them. A good set of debug macros will let you selectively enable and disable different levels of debug verbosity, so that you can immediately go from normal (silent) operation to "full debug" mode just by changing the command line arguments.

There are generally two combinable approaches to specifying what debugging information should be output: Some macros provide a tunable "verbosity level" (e.g., 0-9) and other macros allow you to specify which functionality should be debugged (e.g., socket operations, processes, etc.). More complex systems often allow the user to specify verbosity on a per-component or per-functionality basis ("verbosity 9 process debugging but verbosity 1 socket debugging").

For purposes of 15-441, we suggest using a relatively straightforward set of debug macros that provide one or the other verbosity knobs, but not both. Figure 5.1 shows a simplified version of a debug macro header, debug.h that allows the user to specify the binary-OR of different debug facilities to selectively enable debugging for different components of the program.

5.3 Debugging tools

5.3.1 gdb

`gdb` can run in three ways:

- Starting a new process running under the debugger:

  ```
  gdb binary
  ```

- Examining a `core` file from a crashed process

  ```
  gdb binary core
  ```

- Attaching to a running process

  ```
  gdb binary PID
  ```

For the last form, you can get the PID by using the `ps` command. Often useful is

```
ps auxww | grep your_program_name
```

```
#ifndef _DEBUG_H_
#define _DEBUG_H_

#include "err.h"

#include <stdio.h>  /* for perror */
#ifdef DEBUG
extern unsigned int debug;
#define DPRINTF(level, fmt, args...) \
    do { if ((debug) & (level)) fprintf(stderr, fmt , ##args ); } while(0)
#define DEBUG_PERROR(errmsg)                                          \
    do { if ((debug) & DEBUG_ERRS) perror(errmsg); } while(0)
#define DEBUGDO(level, args) do { if ((debug) & (level)) { args } } while(0)
#else
#define DPRINTF(args...)
#define DEBUG_PERROR(args...)
#define DEBUGDO(args...)
#endif

/*
 * Add some explanatory text if you add a debugging value.
 * This text will show up in -d list
 */

#define DEBUG_NONE      0x00        // DBTEXT:  No debugging
#define DEBUG_ERRS      0x01        // DBTEXT:  Verbose error reporting
#define DEBUG_INIT      0x02        // DBTEXT:  Debug initialization
#define DEBUG_SOCKETS   0x04    // DBTEXT:  Debug socket operations
#define DEBUG_PROCESSES 0x08        // DBTEXT:  Debug processes (fork/reap/e

#define DEBUG_ALL  0xffffffff

#ifdef __cplusplus
extern "C" {
#endif
int set_debug(char *arg);  /* Returns 0 on success, -1 on failure */
#ifdef __cplusplus
}
#endif

#endif /* _DEBUG_H_ */
```

Figure 5.1: An example set of debug macros included via the debug.h file.

Command	Function
	Executing Code
run	Begin executing the program
c	Continue running after breaking
s	Step into next source line, entering functions
n	Run until next source line, stepping over functions
	Getting Information
bt	Display a backtrace of all stack frames
p *expr*	Print the value of *expr*
	e.g., `p (5*5)` or `p *f`
b *n*	Set a breakpoint at line *n*
b *func*	Set a breakpoint at function *func*
list	List the source code at the current point
list *func*	List the source code for function *func*
up	Go up to the previous stack frame
down	Go down to the next stack frame
info locals	Show the local variables for the stack frame

Table 5.1: Common gdb functions.

Why don't I have a core file?

The default system limits may be the culprit here. Type the command `limit`, and see what it says. You should get output that looks like:

```
767 moo:~/systems-se> limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    unlimited    <--- note this line
memoryuse       unlimited
vmemoryuse      unlimited
descriptors     1024
memorylocked    unlimited
maxproc         7168
openfiles       1024
```

Your value of `coredumpsize` may be set to zero, in which case core dumps will be disabled. To fix this, type `unlimit coredumpsize`. Sometimes, you don't *want* core files (because they take time and space to save). You can prevent them by typing `limit coredumpsize 0`.

Different operating systems have different conventions for naming core dump files. Linux may name them simply "core" or "core.PID" (e.g., "core.7337"). BSD names them "program.core". On Mac OS X, the system stores all core dump files in the directory `/cores`.

Many systems prevent core dumps from processes that are setuid or setgid. If you want to get a core from such executables, you'll need to execute them as their owner.

```
#include <stdlib.h>
#include <stdio.h>

int
main()
{
    char *buf;
    buf = malloc(64);
    fprintf(stderr, "accessing start of buf\n");
    buf[0] = '\0';
    fprintf(stderr, "accessing end of buf\n");
    buf[63] = '\0';
    fprintf(stderr, "accessing past end of buf\n");
    buf[64] = '\0';
    fprintf(stderr, "done with overwrite test\n");
    exit(0);
}
```

Figure 5.2: A test program that overwrites a malloc buffer.

Figure 5.3: fig:overwrite

5.3.2 System call tracing: ktrace, strace, and friends

Another good way to debug processes (even if you may not have the source code handy) is to use a system call tracing facility such as `ktrace` (BSD) or `strace` (Linux).

5.3.3 Memory Debugging with Electric Fence

Electric Fence is a utility that helps debug buffer overruns. It monitors access to malloc-generated buffers to detect if your code writes past the end (or start) of the buffer. Running your code under electric fence will be slower, but it's a good way to detect weird, hard to track down errors involving buffer problems.

To run your code under electric fence, run it as:

```
ef <executable> [args]
```

You can also compile your code directly against Electric Fence by linking it with `-lefence`.

Using electricfence, an attempt to access memory outside of the allocated chunk will cause the program to seg fault instead of causing strange, unpredictable behavior.

For example, if you were to type in and run the program in Figure **??** without electric fence, it would (typically) run to completion:

```
546 unix35:~> ./overwrite
accessing start of buf
accessing end of buf
accessing past end of buf
```

```
done with overwrite test
```

However, if you run it under electric fence, it will crash at exactly the instruction that writes outside the malloc buffer:

```
547 unix35:~> ef ./overwrite

  Electric Fence 2.2.0 Copyright (C) 1987-1999
  Bruce Perens <bruce@perens.com>
accessing start of buf
accessing end of buf
accessing past end of buf
/usr/bin/ef: line 20: 11049 Segmentation fault (core dumped)
   ( export LD_PRELOAD=libefence.so.0.0; exec $* )
```

The error message from ElectricFence isn't particularly useful (though it does tell you the process ID that died, which is handy if you're on a system like the Andrew Linux machines that name core dump based on PID), but the resulting core dump file is very valuable:

```
548 unix35:~> gdb ./overwrite ./core.11049
GNU gdb Red Hat Linux (6.1post-1.20040607.43.0.1rh)
...
#0  0x0804847b in main () at overwrite.c:14
14          buf[64] = '\0';
```

5.3.4 Tracing packets with tcpdump and Ethereal

For network projects and debugging distributed systems, packet sniffers such as `tcpdump` and Ethereal can be a life-saver. These utilities can record packets as they go in and out of a machine, and can decode various protocols. `tcpdump` is a more low-level interface to raw packets; it's easy to use from the command line and produces useful output. Ethereal understands many more protocols, has a more powerful set of functions (such as reassembling TCP streams), and has a handy GUI.

5.3.5 Examining output

Being able to do a "diff" of your output is often helpful. In particular, when debugging problems with the second project (file transfers), you may find it very useful to know "What's the difference between the file I tried to send and the file that actually arrived?"

- **Send text-only data.** It may seem obvious, but if you make your data contain only ASCII text, you'll be able to examine it more easily with conventional UNIX command-line tools such as `wc, diff` and `grep`.
- **Use the `cmp` command to compare them.** `cmp` will tell you the byte and line at which the two files first differ.
- **Use the `od` command to view binary data.** `od` will convert your data into a hex or octal dump format. You can then diff the output of this.

- **Open it in an editor.** Emacs and vi will display binary data in a way that will allow you to visually scan for differences. It's not perfect, but as a quick hack, it may help you get over a hump and show you some obvious difference in the files.
- **Examine more powerful diff tools.** Students in the past have had luck using a visual binary diff tool:

  `http://home.comcast.net/~chris-madsen/vbindiff/`

These techniques are not specific to project 2: they're useful in dealing with any system that outputs data.

5.4 Debugging other people's code

Debugging a pile of code—or a binary someone handed you—is harder than debugging your own code. Tools like gdb and system call tracing start to shine here, because they show you what the code was actually doing, instead of forcing you to dig through the code (or an object dump!) to figure out what's up.

Error-avoiding citation: [1]

Bibliography

[1] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000. ISBN 020161622X.